LETTER *Special Section on Foundations of Computer Science — Algorithm, Theory of Computation, and their Applications —*

A Simple Heuristic for Order-Preserving Matching

Joong Chae NA[†], ***Nonmember and*** **Inbok LEE**[††a)], ***Member***

SUMMARY Order preserving matching refers to the problem of reporting substrings in the text which are order-isomorphic to the pattern. In this paper, we show a simple heuristic which runs in linear time on average, based on finding the largest elements in each substring and checking their locations against that of the pattern. It is easy to implement and experimental results showed that the running time grows linearly.

key words: *string algorithm, range minimum query, order preserving matching*

1. Introduction

Order preserving matching refers to the problem of reporting substrings of the text which are order-isomorphic to the pattern. Formally the problem can be defined as follows:

Problem 1: Given a text $T[1,n]$ and a pattern $P[1,m]$ over an integer alphabet, a substring $T[i,i+m-1]$ and P have an *order-preserving matching* if and only if there exists a permutation $\pi = (\pi_1, \pi_2, \cdots, \pi_m)$ on the set $\{1, 2, \cdots, m\}$ such that $P[\pi_1] < P[\pi_2] < \cdots < P[\pi_m]$ and $T[i+\pi_1-1] < T[i+\pi_2-1] < \cdots < T[i+\pi_m-1]$.

Example 1: Assume that $T = (1,3,8,5,2,6,7,9)$ and $P = (11,23,74,43)$. In this case $\pi = (1,2,4,3)$ and we can see that $T[1,4] = (1,3,8,5)$ and P have an order-preserving matching.

The problem is related to time series data analysis. Recently there have been several works on this problem. First the problem was considered in [1] and a KMP-style $O(n + m\log m)$ algorithm was proposed. Order preserving matching with multiple patterns was also considered. In [2], order preserving matching with k mismatches was considered. Hasan et al. [3] studied the order-preserving string regularities. Cho et al. [4] proposed a Boyer-Moore approach on this problem. Chhabra and Tarhio showed a filtration method based on the orders between neighbouring characters in [5]. Kim et al. [6] considered the case when there are ties in strings. Nakamura et al. [7] considered the case where the text is not a simple string, but a tree or a DAG (directed acyclic graph).

Manuscript received March 23, 2018.
Manuscript revised July 27, 2018.
Manuscript publicized October 30, 2018.

[†]The author is with Sejong University, Republic of Korea.
[††]The author is with Korea Aerospace University, Republic of Korea.
a) E-mail: inboklee@kau.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2018FCL0004

2. Algorithm

We first begin with considering the brute-force method. First we sort P and obtain the permutation π in $O(m\log m)$ time. Once we know π, verifying whether two strings $P[1,m]$ and $T[i,i+m-1]$ have an order-preserving matching is straightforward. We will assume that there are no ties in $P[1,m]$ and any substring $T[i,i+m-1]$: they can be broken using the method in [6]. Once we know π, we can check whether $T[i+\pi_1-1] < T[i+\pi_2-1] < \cdots < T[i+\pi_m-1]$ in $O(m)$ time. As there are $n-m+1$ substrings of length m in T, the total time complexity is $O(nm + m\log m)$.

Our algorithm is based on one simple observation.

Observation 1: If two string $P[1,m]$ and $T[i,i+m-1]$ have an order-preserving matching and the smallest (or, largest) element of $P[1,m]$ is $P[j]$, then that of $T[i,i+m-1]$ should be $T[i+j-1]$.

The correctness of Observation 1 is straightforward. We can exploit it as follows. From π, we know that $P[\pi_1]$ is the smallest element of P and $P[\pi_m]$ is the largest one. Therefore, for each $T[i,i+m-1]$, we first check whether $T[i+\pi_1-1]$ is its smallest element (or, whether $T[i+\pi_m-1]$ is its largest one). If so, we check whether $T[i,i+m-1]$ and P have an order-preserving matching using the verification procedure mentioned above. Otherwise, we can safely skip it.

A brute-force implementation of reporting locations of the smallest (or, the largest) elements in all substrings of length m in T will take $O(nm)$ time. But if we use the range minimum query (RMQ) algorithm in [8], it can be done in $O(n)$ time, instead of $O(nm)$. We first preprocess T in $O(n)$ time and for each substring $T[i,i+m-1]$, a range minimum (or, maximum) query asking the location of its smallest (or, the largest, respectively) element takes $O(1)$ time.

We can simplify the process based on the following facts:

- the query range is always of length m,
- we ask only the location of the smallest (or, the largest) element,
- and our queries are always in order from left to right.

We show a simple heuristic here. First we find the largest element of $T[1,m]$. Then, assuming that we know the largest element of $T[i,i+m-1]$, we check whether it is $T[i]$. If so, we find the largest element of $T[i+1,i+m]$

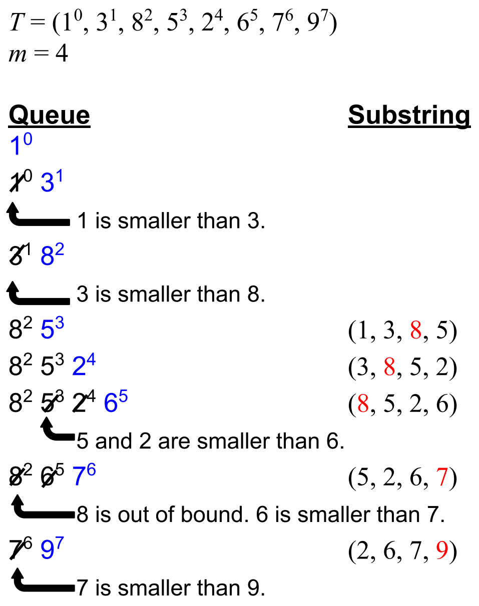

Fig. 1 Reporting the largest elements of substrings of length 4 in $T = (1, 3, 8, 5, 2, 6, 7, 9)$.

by linear scanning. Otherwise, we check whether it is larger than $T[i+m]$. If so, it is still the largest one of $T[i+1, i+m]$. Otherwise $T[i+m]$ is. It takes $O(n)$ time on average and $O(nm)$ in the worst case.

There is a well-known trick which takes $O(n)$ time in the worst case. Here we show how to report the largest elements only, as the same idea can be applied to reporting the smallest ones. We use a deque (double-ended queue) of length m. We maintain that its head element will store the largest element of the substring under consideration. Also, values of elements in the deque are kept in descending order and their indices in T are kept in ascending order.

- **The base case** $T[1, m]$: We first put $T[1]$ into the empty deque. For $T[j]$ $(2 \leq j \leq m)$, we eliminate elements smaller than $T[j]$ from the deque. We scan the deque from the tail to the head and delete such elements, as they cannot be the largest element of any substring. Then we insert $T[i]$ into the deque. It is easy to show that the deque will keep the properties we mentioned above. After handling $T[m]$, the head element is the largest in $T[1, m]$.
- **The step case** $T[i, i+m-1]$ $(2 \leq i \leq n-m+1)$: First we check whether the head element is $T[i-1]$. If so, it is removed from the deque. Then from the tail to head, we again eliminate elements smaller than $T[i]$ and put $T[i]$ at the tail. It is easy to show that the largest elements of $T[i, i+m-1]$ will be at the head of deque.

It is easy to show that whole procedure runs in $O(n)$ time, as each $T[i]$ enters the deque once and elements removed from the deque are not considered again. Figure 1 is an example of reporting the largest elements of substrings of length 4 in $T = (1, 3, 8, 5, 2, 6, 7, 9)$. Indices of elements in T is written next to them.

Now we consider the time complexity. The worst case happens when $P = (1, 2, \cdots, m)$ and $T = (1, 2, \cdots, n)$ and it takes $O(nm)$ time.

Theorem 1: Our heuristic runs in $O(n + m \log m)$ time on average.

Proof. The permutation π can be obtained in $O(m \log m)$ time. For each $T[i, i+m-1]$, the verification is done only when its largest element is $T[i+\pi_1]$. Let X be the number of positions in T where the largest element of some substring of length m in T can appear. Then we will conduct the verification X times and the total number of character comparisons will be equal to or smaller than $X \cdot m$, in addition to $O(n)$ comparisons to find the largest elements.

Let $X_1, X_2, \cdots, X_{n-m+1}$ be indicator variables where $X_1 = 1$ and $X_{i+1} = 0$ if the index of the largest element in $T[i, i+m-1]$ is the same with that of $T[i+1, i+m]$ and $X_{i+1} = 1$ otherwise. Then, it is easy to show that

$$X = X_1 + X_2 + \cdots + X_{n-m+1}$$

is the number of distinct positions in T that can correspond to the largest elements of some substring of length m.

Now we calculate $E[X_{i+1}]$. There are two cases where $X_{i+1} = 1$:

- The maximum value of $T[i, i+m-1]$ was $T[i]$, or
- that of $T[i+1, i+m]$ was at $T[i+m]$.

Therefore, $E[X_{i+1}] = Pr(X_{i+1} = 1) = 1/m + 1/m - 1/m^2$. By linearity of expectation, we get

$$E[X] = 1 + \sum_{i=2}^{n-m+1} E[X_i] = 1 + (n-m)\left(\frac{2}{m} - \frac{1}{m^2}\right) \leq \frac{2n}{m}.$$

From the fact that the number of character comparisons is at most $X \cdot m$, we get

$$E[X \cdot m] \leq \frac{2n}{m} \cdot m = 2n.$$

Therefore, the expected number of character comparison is $O(n)$. □

3. Experimental Results

We implemented our algorithm in C++ and performed on a iMac machine with Intel Core i5 processor running macOS 10.13.3 and 4G RAM.

The first observation is that while the well-known trick for finding the maximum elements of all the substrings of length m runs in linear time, it is slower than the simple heuristic in practice. Even though it may run in $O(nm)$ time, we think that the probability of facing the worst case is negligible for random texts and that the burden of maintaining the deque is heavy for short patterns. Therefore we used the simpler heuristic here.

We compared the performance of our heuristic against

Table 1 Search time (in seconds) with a random text of length 1,000,000 and 1,000 random patterns. The alphabet size is 10,000.

	$m = 5$	$m = 10$	$m = 15$	$m = 20$
Ours	13.59	11.21	10.48	9.85
Filtration [5]	13.12	12.20	12.66	11.85
q-gram [4]	13.71	7.58	6.40	6.86

Table 2 Search time (in seconds) with a random text of length 500,000 and 1,000 random patterns. The alphabet size is 10,000.

	$m = 5$	$m = 10$	$m = 15$	$m = 20$
Ours	6.82	5.30	5.32	4.97
Filtration [5]	6.17	5.39	5.96	5.91
q-gram [4]	7.14	3.19	3.30	3.03

two previous algorithms. One is based on the Horspool algorithm with q-grams [4] and the other is based on filtration [5]. The source code of [4] was kindly provided by its authors and we implemented the algorithm of [5] by ourselves. Rabin-Karp fingerprinting was used to check the equality of two binary strings.

Table 1 and 2 show the experimental results with random texts and random patterns. Note that all three algorithms are independent of the size of the alphabet. As the length of the pattern grows bigger, the search time is reduced for all three algorithms we considered. In our algorithm, the number of changes in the maximum elements will be reduced with longer patterns. When $m = 5$, the q-gram approach suffered from small shifts. For all the other cases it was the fastest. As our heuristic is also based on filtration, it is fair to compare its performance against that of [5]. Overall they showed similar search time but our implementation of the algorithm in [5] may not be the same with that of its authors.

4. Conclusion

We presented a simple heuristic for order-preserving matching based on finding the smallest (or, the largest) elements in substrings of length m. We showed that the expected running time is $O(n)$. The proposed heuristic is easy to understand and to implement.

Experimental results show the performance is on par with the filtration approach in [5]. Also, one may devise a faster hybrid heuristic by combining ours with previous ones [4], [5]. We believe that our idea can be applied to variations of order preserving matching, including multiple pattern matching and wild card matching.

Acknowledgments

Joong Chae Na was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2017R1D1A1B03029451).

References

[1] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C.S. Iliopoulos, K. Park, S.J. Puglisi, and T. Tokuyama, "Order-preserving matching," Theor. Comput. Sci., vol.525, pp.68–79, 2014.
[2] P. Gawrychowski and P. Uznański, "Order-Preserving Pattern Matching with k Mismatches," CPM 2014, pp.130–139, 2014.
[3] M.M. Hasan, A.S.M.S. Islam, M.S. Rahman, and M.S. Rahman, "Order preserving pattern matching revisited," Pattern. Recogn. Lett., vol.55, pp.15–21, 2015.
[4] S. Cho, J.C. Na, K. Park, and J.S. Sim, "A fast algorithm for order-preserving pattern matching," Inf. Process. Lett., vol.115, no.2, pp.397–402, 2015.
[5] T. Chhabra and J. Tarhio, "A filtration method for order-preserving matching," Inf. Process. Lett., vol.116, no.2, pp.71–74, 2016.
[6] J. Kim, A. Amir, J.C. Na, K. Park, and J.S. Sim, "On Representations of Ternary Order Relations in Numeric Strings," Mathematics in Computer Science, vol.11, no.2, pp.127–136, 2017.
[7] T. Nakamura, S. Inenaga, H. Bannai, and M. Takeda, "Order Preserving Matching on Trees and DAGs," SPIRE 2017, pp.271–277, 2017.
[8] M.A. Bender and M. Farach-Colton, "The LCA problem revisited," LATIN 2000, vol.1776, pp.88–94, 2000.